Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-93-15

1993-01-01

A Characterization of the Computational Power of Rule-based Visualization

Kenneth C. Cox and Gruia-Catalin Roman

Declarative visualization is a paradigm in which the process of visualization is treated as a mapping from some domain (typically a program) to an image. One means of declaring such mappings is through the use of rules which specify the relationship between the domain and the image. This paper examines the computational power of such rule-based mappings. Computational power is measure using three separate criteria. The first of these uses the Chomsky hierarchy, in which computational power is treated as string-acceptance; with this criterion we are able to show that certain rule-based models are equivalent in power to Turing...
Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Cox, Kenneth C. and Roman, Gruia-Catalin, "A Characterization of the Computational Power of Rule-based Visualization" Report Number: WUCS-93-15 (1993). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/302

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/302

A Characterization of the Computational Power of Rule-based Visualization

Kenneth C. Cox and Gruia-Catalin Roman

Complete Abstract:

Declarative visualization is a paradigm in which the process of visualization is treated as a mapping from some domain (typically a program) to an image. One means of declaring such mappings is through the use of rules which specify the relationship between the domain and the image. This paper examines the computational power of such rule-based mappings. Computational power is measure using three separate criteria. The first of these uses the Chomsky hierarchy, in which computational power is treated as string-acceptance; with this criterion we are able to show that certain rule-based models are equivalent in power to Turing machines. The second criterion is the evaluation of recursive functions, while the third is a more informal consideration of the abstractive capabilities of the mapping.

School of Engineering & Applied Science

A Characterization of the Computational Power of Rule-based Visualization

Kenneth C. Cox
Gruia-Catalin Roman

WUCS-93-15

February 1993

To appear in the Journal of Visual Languages and Computing.

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Printed 3/23/93

Abstract

Declarative visualization is a paradigm in which the process of visualization is treated as a mapping from some domain (typically a program) to an image. One means of declaring such mappings is through the use of rules which specify the relationship between the domain and the image. This paper examines the computational power of such rule-based mappings. Computational power is measured using three separate criteria. The first of these uses the Chomsky hierarchy, in which computational power is treated as string-acceptance; with this criterion we are able to show that certain rule-based models are equivalent in power to Turing machines. The second criterion is the evaluation of recursive functions, while the third is a more informal consideration of the abstractive capabilities of the mapping.

Keywords: declarative visualization, computational power, rule-based mappings

Correspondence: All communications regarding this paper should be addressed to

Dr. Gruia-Catalin Roman
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

office: (314) 935-6190
secretary: (314) 935-6160
fax: (314) 935-7302

roman@cs.wustl.edu

1. Introduction

Visualization is the representation of symbolic information in graphical form. This definition encompasses a wide range of applications, such as *scientific visualization* (the presentation of data about physical processes, whether obtained from instruments or from simulations), *visual programming* (the display of programs in graphical form, normally coupled with facilities to modify the structure by changing the graphics), and *program visualization* (the presentation of information about the execution of programs), the subject of this paper.

Program visualization has become increasingly popular in recent years, with applications ranging from simple monitoring of program execution to pedagogical presentations of algorithms designed to enhance the viewer's understanding. As we have argued previously [6], program visualization may be viewed as a mapping from some aspect of the program (code, data or control state, or execution behavior) to a graphical representation. This *declarative* treatment of visualization is used indirectly by such traditional systems as Balsa and Zeus [1,2], where program events are mapped to operations on an image. Other systems, such as Aladdin [4], Tango [9], and our own Pavane [7] are more explicit in their modeling of visualization as a mapping.

The mappings used in program visualization range from simple direct associations between the values of program variables and the attributes of graphical objects to complex transformations of arbitrary data structures into multi-part graphical representations. In general, simpler mappings are to be preferred since they are more amenable to efficient (and even automatic) implementation. A reasonable question then becomes, are such mappings powerful enough to construct the types of images needed in program visualization?

This paper answers this question by presenting an analysis of the capabilities of declarative mappings. This work was motivated by our own research with Pavane, and is greatly assisted by Pavane's formal foundation (anchored in the predicate and set calculus). However, the analytic techniques presented here could also be used with other declarative systems, and we believe that many of our results could be immediately applied to such systems.

Pavane treats visualization as a mapping from the state of a program (called the *underlying computation*) to a collection of graphical objects with time-variant attributes. The mapping is specified by means of rules which resemble those of production systems. Conceptually, each time the state changes all the rules are re-evaluated to produce a new collection of graphical objects which is then displayed. In our work, we have identified three distinct ways in which such rule evaluations can be performed. We call these the *linear model* (in which no information about previous evaluations is retained), the *history model* (in which the results of the most recent evaluation are available to the rules, which are applied once), and the *fixpoint model* (in which the rules are repeatedly applied until their output stabilizes). Our results show that these models all have different computational power, whether that power is defined in terms of the Chomsky hierarchy of languages and automata, the evaluation of a common type of recursive function, or the notion of "abstractness" of the mapping.

The remaining sections of this paper present our results in greater detail. Section 2 introduces the rule-based paradigm and the three evaluation models. Section 3 presents a firm theoretical characterization of the power of these rule-based models in terms of finite automata. Section 4 returns to program visualization by considering a common problem in the construction of images—the evaluation of state-dependent recursive functions—and the capabilities of the rule-based models to solve this problem. Section 5 is a less-formal consideration of the abstractive capabilities of the models in terms of an abstraction hierarchy. Section 6 summarizes the results and presents some conclusions.

2. Rule-Based Visualization

Pavane treats visualization as a mapping from the state of a program called the *underlying computation* to an image. The mapping is specified by means of rules which resemble those of production systems. Pavane also uses a tuple-based representation of data like that of other rule-based systems. A tuple consists of a *type* (an arbitrary name) and a number of components; the number of components associated with a particular type, called the *arity* of the type, is fixed, and the components may be identified by names. Tuples may be denoted by the syntax:

$$type(value_1 , value_2 , \ldots value_n)$$

which represents a tuple of the indicated *type* whose *n* components have the indicated *values*: The assignment of values to components is determined by the position within the tuple, so all *n* components of an arity-*n* tuple must be present. An alternate notation uses the names of the components to unambiguously assign values:

type (name1 := value1 , name2 := value2 , ...)

In this case the values for certain components may be omitted and are assumed to take on default values.

In Pavane, sets of tuples called *spaces* are used to describe both the state of the underlying computation and the final image. Tuple-based representation of state has a number of attractive features. The tuples provide a uniform representation of both data and control state which is compatible with many other representations of programs. For example, in a computational model which uses variables the fact that "*x = 5*" can be represented by the presence of a tuple *x(5)* in the state space (we normally use the first of the notations given above for tuples in the state). If the value of *x* later changes to 6, the tuple *x(5)* is removed from the state space and the tuple *x(6)* is added. Similarly, tuples can capture control information at any desired level from a record of processor program counters to a simple indication of active/inactive status.

We similarly represent the range of the mapping—the image—as a set of *animation tuples* called the *animation space*. Each animation tuple represents a single graphical object positioned in a three-dimensional coordinate system. We normally use the second of the notations given above for animation tuples, with the type of the graphical object (line, sphere, etc.) used as the type of the tuple and the various attributes of the graphical object used as the names of the components. For example, a sphere centered at coordinate [0,0,10] of radius 0.5 could be represented by the tuple *sphere(center := [0,0,10], radius := 0.5)*. Unspecified attributes (e.g., this sphere's color) take on a default value.

In the simplest case, the visualization maps the state space directly to the animation space as shown in the top portion of Figure 1. The mapping is expressed as a collection of rules. Syntactically a rule has the form:

```
rule_name ≡
        variables :
                query
        ⇒
                production ;
```

The *rule_name* serves only for identification purposes. The *variables* are identifiers. The *query* is an arbitrary predicate which may include tests for the presence or absence of tuples in the state space, while the *production* is a list of animation tuples. Both the *query* and the *production* may use the *variables*. An informal, operational interpretation of such a rule is, "Given the state space, for every instantiation of the *variables* such that the *query* is true, place the corresponding tuples of the *production* in the animation space." The animation space produced by the mapping is simply the collection of all the tuples produced by all the rules. We assume that each time the state space changes, the rules are evaluated to determine the animation space, which is then translated into an image.

The middle portion of Figure 1 shows the code of a complete Pavane visualization. The visualization starts with the line *"state ⇒ animation"* indicating that this mapping directly transforms the state space to the animation space. The tuple types used in the state space are then declared (the tuples of the animation space, representing graphical objects, are fixed by the system). The state of this computation consists of tuples of the type *process(id,status)*, with both *id* and *status* integers. The two rules of this visualization are next declared. The first rule, *draw_active*, generates a filled circle for each *process* tuple with *status* greater than 0—or more technically, for each instantiation of the variables *pid* and *status* such that there is a *process(pid,status)* tuple in the state space and *status > 0*, a *circle(center := [pid,0,0], radius := 0.3, fill := true)* tuple is present in the animation space. The second rule is similar but generates unfilled circles. The bottom portion of the figure shows a possible state space, the corresponding animation space, and the image represented by this animation space.

Our original concept of a rule-based declarative system [6] only permitted such direct mappings from the state to the image. Subsequent development work showed that these mappings are both inconvenient to use and incapable of producing many of the visualizations that we wanted. This led to the development of three distinct models for rule-based visualization, described below.

state space *animation space*

(a)

```
visualization simple_example
        state ⇒ animation;
state space
        ⟨ integer pid, status :: process(pid,status) );
rule draw_active ≡
        integer pid, status :
                process(pid,status), status > 0
        ⇒
                circle(center := [pid,0,0], radius := 0.3, fill := true);
rule draw_inactive ≡
        integer pid, status :
                process(pid,status), status ≤ 0
        ⇒
                circle(center := [pid,0,0], radius := 0.3, fill := false);
end
```

(b)

State space	*Animation space*
process(0,5)	circle(center := [0,0,0], radius := 0.3, fill := true)
process(1,1)	circle(center := [1,0,0], radius := 0.3, fill := true)
process(2,-3)	circle(center := [2,0,0], radius := 0.3, fill := false)
process(3,1)	circle(center := [3,0,0], radius := 0.3, fill := true)
process(4,0)	circle(center := [4,0,0], radius := 0.3, fill := false)
process(5,0)	circle(center := [5,0,0], radius := 0.3, fill := false)
process(6,2)	circle(center := [6,0,0], radius := 0.3, fill := true)

(c)

Figure 1. Simple example of a Pavane rule-based mapping directly from the state space to the animation space. The mapping transforms process status information represented by *process* tuples into a row of circles. (a) Illustration of the form of the mapping. (b) The complete Pavane program. (c) Sample state space, corresponding animation space, and image.

2.1. Linear Model

It is often inconvenient to construct visualizations using the direct mapping from the state space to the animation space that was described above. For this reason, we permit the overall mapping to be decomposed into an arbitrarily-long (but fixed-length) sequence of stages as illustrated in Figure 2. We refer to this type of mapping as the *linear model*, referring to the linear arrangement of the stages.

The intermediate spaces of these mappings are also sets of tuples, with space names and tuple type-names chosen by the animator; for ease of compilation, we require that each tuple type-name belong to only one space in the mapping. Each transformation between two consecutive spaces in the sequence is itself a mapping composed of rules as described above. We use the generic terms *input space* and *output space* in connection with these mappings; the query part of the rules may test for the presence or absence of tuples in the mapping's input space, and the production is a list of tuples in the output space. When the state space changes, each of the sub-mappings is evaluated in order to produce its output space, which then serves as the input space of the next mapping. The input space for the first mapping is the state space of the underlying computation, while the output space of the last mapping in the sequence is the animation space. The reader will recognize that the direct mapping presented above is simply a one-stage linear mapping.

Figure 2 gives the rules for a simple linear mapping which performs the same transformation as the mapping in Figure 1. An intermediate space containing tuples of the types *active_proc* and *inactive_proc* is used in this mapping. The first two rules map from the state space to this intermediate space and the second two from the intermediate space to the animation space. The lower portion of the figure shows a state space, the corresponding intermediate and animation spaces, and the image.

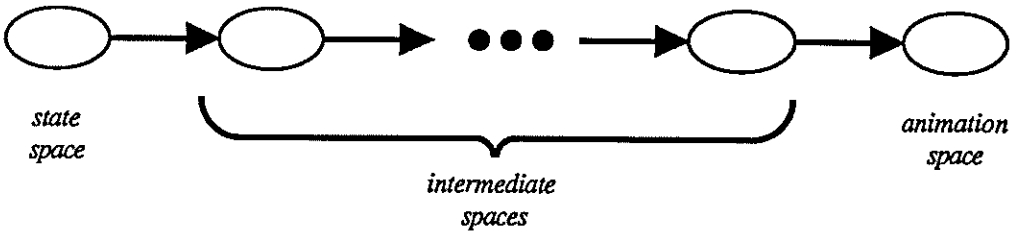
The mathematical formalization of the linear model (which we do not present here) indicates, as might be expected, that the animation space is completely determined by the state space. A somewhat more surprising result is that intermediate spaces are not necessary in the linear model—anything that can be computed by a multi-stage mapping can be computed by a single-stage mapping directly from the state space to the animation space, as exemplified by the relation between the mappings in Figures 1 and 2. The transformation from a given multi-stage mapping to a single-stage mapping is completely mechanical and little more than an exercise in the predicate calculus.

2.2. History Model

Although the linear model permits the construction of a wide variety of visualizations, it has several significant shortcomings. Chief among these is the fact that production of certain types of effects is difficult or impossible. Consider as a simple example a sorting algorithm which operates on an array of numbers. The algorithm operates by selecting two elements which are not in sorted order and exchanging them, and we wish to show this exchange operation by a corresponding motion of the graphical objects representing the array elements. Such a depiction is not possible in the linear model, since it requires knowledge of both the current and previous state of the computation; yet such effects (which we call *visual events*) are very useful, even essential, to many visualizations.

This consideration led us to develop what we call the *history model*, illustrated in Figure 3. In this model, the overall mapping is again decomposed into a series of sub-mappings. The system maintains both the current and the previous versions of each of the spaces. Rules that contribute to a particular output space are permitted to examine the current and previous versions of the rule's input space as well as the previous version of the output space, as shown in the top right of the Figure. The prefix *"old."* is used on tuples to indicate that the previous version of the space should be examined. We also permit the animator to specify the initial contents of the intermediate spaces; these are then used as the previous space for the first evaluation of the rules. If no such initialization is given for a space, the previous space is empty on the first evaluation of the rules.

Operationally, evaluation of the rules in the history model begins when the underlying computation produces a new state space. Each of the sub-mappings are evaluated in sequence as in the linear model, using the results of the previous evaluation of each stage as the previous space. The mathematical formalism for the mappings can be easily extended to incorporate the previous spaces. From this formalism, we find (obviously) that the animation space is no longer dependent only on the state space, but also on the previous versions of the state and



(a)

```
rule find_active ≡
        integer pid, status : process(pid,status), status > 0
            ⇒ active_proc(pid);

rule find_inactive ≡
        integer pid, status : process(pid,status), status ≤ 0
            ⇒ inactive_proc(pid);

rule draw_active ≡
        integer pid :active_proc(pid)
            ⇒ circle(center := [pid,0,0], radius := 0.3, fill := true);

rule draw_inactive ≡
        integer pid : inactive_proc(pid)
            ⇒ circle(center := [pid,0,0], radius := 0.3, fill := false);
```

(b)

State space	*Intermediate space*	*Animation space*
process(0,5)	active_proc(0)	circle(center := [0,0,0], radius := 0.3, fill := true)
process(1,1)	active_proc(1)	circle(center := [1,0,0], radius := 0.3, fill := true)
process(2,-3)	inactive_proc(2)	circle(center := [2,0,0], radius := 0.3, fill := false)
process(3,1)	active_proc(3)	circle(center := [3,0,0], radius := 0.3, fill := true)
process(4,0)	inactive_proc(4)	circle(center := [4,0,0], radius := 0.3, fill := false)
process(5,0)	inactive_proc(5)	circle(center := [5,0,0], radius := 0.3, fill := false)
process(6,2)	active_proc(6)	circle(center := [6,0,0], radius := 0.3, fill := true)

(c)

Figure 2. A linear mapping, identical in function to that of Figure 1, which uses two stages. (a) The general form of a linear mapping. (b) The rules used by the mapping; the tuple types *active_proc* and *inactive_proc* are members of the intermediate space of the mapping. (c) Sample state space, corresponding intermediate and animation spaces, and image; the animation space, and hence the image, are the same as that of Figure 1.

intermediate spaces. Further, unlike the linear model, single-stage history mappings are *not* equivalent to multiple-stage mappings; there are certain constructs that can only be generated using multiple stages.

The history model allows us to produce many of the desired visual events. Figure 3 presents a rule which detects and depicts the exchanges involved in the sorting example. This rule also illustrates Pavane's (rather primitive) facilities for producing visual effects such as smooth movements or size changes. Rather than representing a single, static image, the animation tuples are considered to represent a *sequence* of images called *frames* which are identified by natural number "times". The attributes of graphical objects may be assigned functions whose value implicitly depends on this time, such as the *ramp* function used in the figure. The value of this function,

ramp(2,[j,0,0],10,[i,0,0])

is *[j,0,0]* at time (frame number) less than or equal to 2, *[i,0,0]* at time greater than or equal to 10, and a linear interpolation between these two values at intermediate times. The program which translates the animation tuples into images determines the maximum time of any of these functions and produces a sequence of frames, one for each number up to the calculated maximum, with the values of the object attributes computed from the functions. The image in the figure shows one such frame produced during the exchange (the other rectangles and the arrow are produced by additional rules which are not shown).

Although our original motivation for the history mapping was to permit visual events (and indeed, in one early version of Pavane the history could be used only in the mapping that produced animation tuples), we quickly discovered a number of other applications. An obvious use is in the depiction of the history of the program's behavior; information about earlier program actions can be saved using the history capability, then transformed into an image. Similar applications, discussed in greater detail in Section 4, involve the use of the history space to incrementally calculate geometric information. Finally, the history capability is very useful when constructing visualizations that illustrate particular formal properties of the program (a technique we call *analytical representation*, or *proof-based visualization*), since these formal properties often refer to previous program activities.

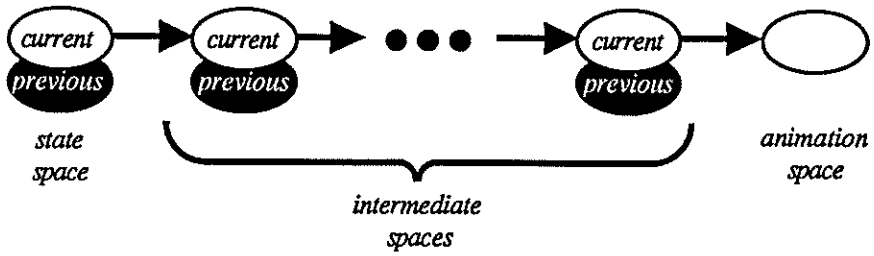
2.3. Fixpoint Model

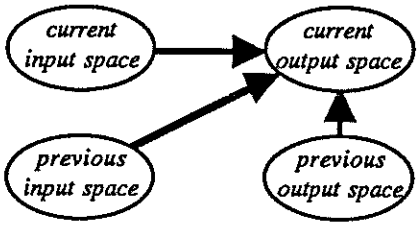
Our third model of rule-application was prompted by the discovery that the history model is not capable of producing certain results. The specific problem that we encountered (described in greater detail later in Section 4) was the visual representation of a tree data structure. We found that, unless the underlying computation behaved in a very restricted manner, the history model was not able to compute image coordinates for the nodes of the tree. Many other common calculations (mostly, but not entirely, connected with the computation of geometric information) were also not computable, which is obviously unacceptable in a visualization system.

The difficulty was shown to arise from the unbounded nature of the computations. Specifically, the problems required an unbounded *serial* computation, that is, a computation in which a value is computed, and the result used to compute a second value, and so on indefinitely. The only way in which the linear and history models can perform this type of computation is through the use of multiple sub-mappings, and the number of these must be fixed; hence, unbounded serial computation is not possible in these models. (As an aside, unbounded *parallel* computation, that is, of multiple independent values, is possible and indeed is implied by the informal description of rule semantics: "Find *all* instantiations of the variables such that the query is true of the input space, and add the corresponding tuples to the output space." The process of finding all instantiations introduces the parallelism.)

The problem was thus to introduce a mechanism whereby unbounded serial computation could be introduced into the rule-based approach. We considered several possibilities and eventually decided to use *fixpoint application* of the rules. The fixpoint model uses the same basic organization as the history model, i.e., a linear series of spaces with both the current and previous spaces maintained, as shown in Figure 4. Some or all of the rules are indicated as fixpoint rules using the keyword *fixpoint* before the rule. Sub-mappings that do not contain such rules are evaluated in the same manner as in the history model.

Sub-mappings that contain fixpoint rules are evaluated differently. In these mappings, the previous output space is discarded—or more technically, when the mapping is first applied, the previous version of the output space is treated as empty. All the rules in the mapping are evaluated once, generating an output space. This output space is then taken as the previous output space, and the fixpoint rules (only) of the mapping are re-applied, producing

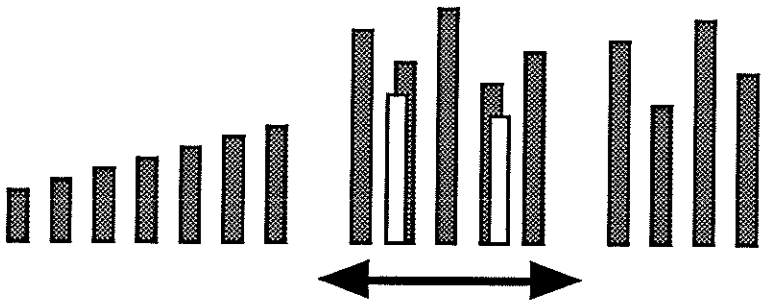




(a)

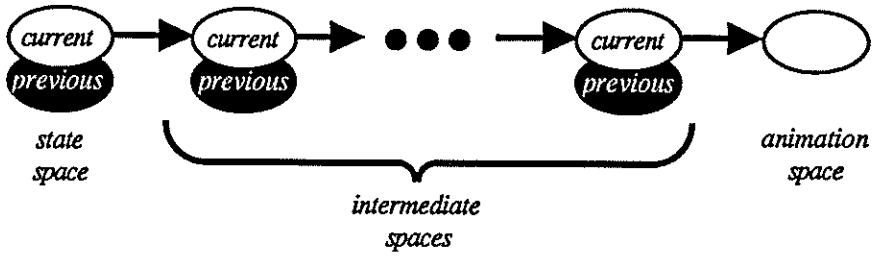
```
rule draw_exchanges ≡
        integer i, vi, j, vj :
                array_element(i,vi), array_element(j,vj), i ≠ j,
                old.array_element(i,vj), old.array_element(j,vi)
        ⇒
                rectangle(corner := ramp(0,[j,0,0],10,[i,0,0]), xsize := 0.5, ysize := vi),
                rectangle(corner := ramp(0,[i,0,0],10,[j,0,0]), xsize := 0.5, ysize := vj);
```

(b)



(c)

Figure 3. Illustration of the history model. (a) In the history model, the most recent version of each space is maintained. The rules that generate an output space may examine the current and previous versions of the input space and the previous version of the output space. (b) A rule designed for a sorting algorithm which uses the history capability to identify and animate the exchanges of elements. (c) A frame from the animation sequence that this rule (in conjunction with others) might produce. The two white rectangles are being swapped between the positions indicated by the arrows.



(a)

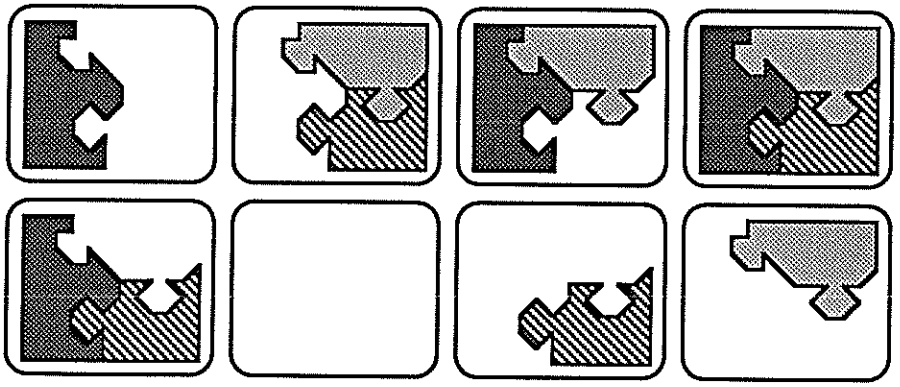
```
fixpoint rule generate_es ≡
        true ⇒ subset(∅);

fixpoint rule generate_subset ≡
        set of integer T; integer a : old.subset(T), element(a)
                ⇒ subset(T ∪ {a});
```

(b)

Input space	*First application*	*Second application*	*Third application*
element(0)	subset(∅)	subset(∅)	subset(∅)
element(1)	subset({1})	subset({1})	subset({1})
element(2)	subset({2})	subset({2})	subset({2})
	subset({3})	subset({3})	subset({3})
		subset({1,2})	subset({1,2})
		subset({1,3})	subset({1,3})
		subset({2,3})	subset({2,3})
			subset({1,2,3})

(c)



(d)

Figure 4. Illustration of the fixpoint model. (a) The basic structure is the same as that of the history model. (b) Fixpoint rules to generate all subsets of a given set. (c) An input space and the sequence of output spaces produced by application of the rules. The last space is the fixpoint (re-evaluation of the rules produces the same space). (d) These rules might be used to generate all possible combinations of some group of objects.

another output space. This process of re-applying the fixpoint rules continues until the current and previous output spaces are identical, in other words, until a fixpoint of the rules is reached. (As shown in the next section, given an arbitrary mapping the question of whether or not it reaches a fixpoint is undecidable. However, we assume that the animator constructs the rules to guarantee a fixpoint, in much the same way that we assume the traditional programmer writes procedures that terminate.)

Although the process as described seems complex, it has a rather simple mathematical formalism expressible, as might be expected, in terms of fixpoints of functions. Further, the fixpoint computation can be implemented with reasonable efficiency using an incremental, RETE-like algorithm [3] which only processes the *changes* in the output space. Finally, the use of the fixpoint provides us with the needed capability to perform unbounded serial computation—indeed, as we will show in the next section, the fixpoint model is computationally equivalent to a Turing machine.

We illustrate the fixpoint model with the rules in Figure 4, which compute the power set of a given set S. The set is provided in the input space for these rules in the form of *element(x)* tuples, one for each element x of S. The rules compute an output space containing *subset(T)* tuples, one for each subset T of S. The first rule generates the empty set as a subset of S, while the second rule states that if we can find a (previously-computed) set T that is a subset of S and a is an element of S, then we should produce $T \cup \{a\}$ as a subset of S. We can demonstrate that when these rules reach a fixpoint, the output space will contain a tuple *subset(T)* if and only if T is a subset of the original set S. This power set example is somewhat contrived, but something similar might be used where we wished to depict all possible combinations of some graphical elements, as shown in the bottom portion of the figure.

3. Finite Automata and Language Recognition

In this section, we present a formal characterization of the capabilities of the three rule-based mapping models described above. Our results are obtained using automata theory, and more specifically the string-acceptance problem, a standard means of evaluating the power of a proposed computational model. The theory behind the characterization of computational power was developed by many researchers over a period of decades. We assume that the reader has a basic familiarity with this fundamental work and only summarize a few key results.

A *finite automaton* is a simplified model of a computer, such as a finite state machine (FSM) or Turing machine (TM). These automata have an internal state (one of a finite set of states) and can manipulate symbols from a finite alphabet. The automaton may have an infinite external storage capacity, such as the TM's tape, but can only observe and modify a finite portion of this store at any time. Computation is modeled as a series of discrete steps in which the automaton, based on its state and the symbols it observes, changes state, alters the symbols it is observing, and/or changes which portion of its storage it is examining. The total amount of computation (defined as number of state changes, symbol modifications, and observation modifications) that the automaton performs in a single step must be finite and bounded. Unbounded computation can be performed only by performing an unlimited number of automaton steps.

One problem which is commonly used to characterize the power of machines is *language recognition*. Given a finite set Σ of symbols, the set Σ^* is defined as the set of all finite sequences of symbols from Σ. A language over Σ is a subset of Σ^*, that is, a (possibly infinite) set of strings. The language recognition problem is generally presented in the form of *string acceptance:* Given a language L and a string ω, is ω contained in L? Investigation of the abilities of finite automata to solve string acceptance problems led to the discovery of a correspondence between the Chomsky grammar-based language classification and the finite automata. Specifically, it was shown that the sets of languages generated by particular types of grammars were identical to the sets recognized by particular types of automata. The resulting classification is generally called the Chomsky hierarchy:

Language Class	Chomsky Grammar	Recognizing Automata
Recursively enumerable	Type 0	Turing Machine (TM)
Context-sensitive	Type 1	Linear-bounded TM
Context-free	Type 2	Nondeterministic Push-Down Automata (NDPA)
Regular	Type 3	Finite State Machine (FSM)

The development of the hierarchy has led to a fairly standard way of evaluating the power of a proposed computational model by demonstrating that it is equivalent to one of the automata in the hierarchy. This is generally done by *emulation*; a method whereby the proposed model can mimic the selected automaton is shown, thus demonstrating that the model is at least as powerful as the automaton.

3.1. String-Acceptance as a Rule-Based Mapping Problem

We will use the string-acceptance problem to formally characterize the computational capabilities of rule-based visualization. Although string-acceptance may seem rather remote from program visualization, it is related to a common type of problem. Consider the symbols of the string as representing various actions that the underlying computation can take. The string as a whole then represents the history of the computation, and string acceptance can be seen to be the recognition of legal (or illegal) behaviors. As a simple example, the P and V operations that have been performed on a semaphore must at all times be described by the regular expression $((PV)^* \cup ((PV)^*P))$ (alternating P's and V's). We could also cite stack operations; considering a push operation as a '(' symbol and a pop as a ')', the sequence of operations must at all times be a prefix of a string in which the parentheses balance. Clearly, we might wish provide some visual indication of a violation of either of these conditions.

We must first consider how to cast string-acceptance into a form suitable for the rule-based approach. With finite automata, the string is presented to the automaton (either one character at a time, for the FSM and NDPA, or in its entirety, for the TM) and the automaton must either accept or reject the string as a member of the target language. We can use a similar approach in the rule-based model; the string will be presented in the *initial space*, and the mapping must produce in the *final space* a tuple of the type *accept(b)*, where *b* is either *true* or *false* depending on whether the string is part of the language. We will permit the mapping from the initial space to the final space to be decomposed into a sequence of mappings. The presentation of the string can be done in two ways, leading to separate problems:

1) The string is presented one symbol at a time, in the form of a *character(c)* tuple in the initial space. The mapping is evaluated once for each symbol. We call this the *incremental string acceptance problem* (ISA).

2) The string is presented in its entirety, in the form of an *omega(s)* tuple in the initial space. The mapping is evaluated exactly once. We call this the *global string acceptance problem* (GSA).

As might be expected, the ISA and GSA lead to different characterizations of the computational power of the models, since in ISA the mapping is evaluated once per symbol in the string and in GSA it is evaluated only once. It is therefore to be expected that we can emulate some automata for the ISA problem that we cannot emulate for the GSA. We also point out that ISA corresponds to the presentation of input to a FSM on a read-once-only input tape, while GSA is similar to the presentation of input to a TM on the TM's tape.

One additional issue must be considered. When attempting to show that a proposed computational model (in our case, rule-based visualization) is equivalent to some finite automaton, certain restrictions need to be made on the types of operations that the proposed model can perform. These restrictions ultimately stem from the basic assumptions about computation embodied in the development of finite automata, namely that the model is capable of manipulating only a finite number of symbols and in each step can perform only a finite and bounded (hereinafter *FB*) amount of computation.

One effect of the restrictions is on the manipulation of integers. It is perfectly permissible for a model to manipulate a finite set of integers (conceptually, each integer is represented as one of the symbols in the machine alphabet), but no automaton is allowed to manipulate *arbitrary* integers in a single step. Such integers can be processed, but only by recording them in the external storage and manipulating the stored information over a series of steps. A similar consideration applies in the manipulation of other data types. Consider for example a sequence (a list, string, or similar data type). We can assume that testing if such a sequence is empty, computing its head (first element), computing its tail (remainder after removing the head), and prepending an element to the list (the LISP "cons" operation) are all FB. With this assumption, a number of other operations are also FB, such as computing the result of applying the tail operation a fixed number of times. Many other operations are not FB. For example, determining if some symbol appears anywhere in the sequence is not permissible—since the length of the sequence is unbounded, the search requires unbounded time.

With these requirements in mind, we propose to restrict the form of our mappings. These restrictions will guarantee that application of the rules meets the FB requirements of the finite automata. We then demonstrate that we can emulate the operation of various automata, thus showing that the rule-based model is *at least as powerful* as the automata. The restrictions apply to the mapping as a whole, the contents of the mapping spaces, and the individual rules. We list each restriction below with a brief explanation of how it contributes to maintaining the overall FB requirement. (Note that some of these "restrictions", particularly the first two, are always required of our rule-based mapping; we include them only to reinforce the argument that the entire process obeys the FB requirement.)

- The number of sub-mappings (and hence intermediate spaces) must be finite and constant, as must the number of tuple types in each intermediate space. This guarantees that application of the mapping as a whole is FB provided application of each sub-mapping is FB.

- The number of rules in any sub-mapping must be finite and constant. This guarantees that application of the sub-mapping is FB provided application of a single rule is FB.

- The rules may manipulate symbols taken from finite sets, as well as lists of such symbols, using the FB sequence operations *head*, *tail*, *cons*, and *isempty* described above. The use of integers and quantified expressions (universals and existentials) is disallowed. This guarantees that performance of all the individual operations appearing in a rule is FB.

- The query portion of the rule must be of fixed, finite length and consist only of tests for the presence or absence of tuples and of predicates using the operations described above. Similarly, the production portion of the rule must be of fixed, finite length. This guarantees that application of a rule is FB provided that the tuple space operations are all FB.

- The number of tuples of a particular type stored in a space must at all times be finite and bounded. This ensures that tuple space operations (searching, insertion, and deletion) are all FB.

From the above list, it can be seen that a single application of the rules of a mapping will be FB. We therefore define such an application as a "step" in the rule-based model. Note that this is *not* the same as saying that a step is a single *evaluation* of the mapping. In the linear and history models, a step is equivalent to evaluation of the mapping; however, in the fixpoint model an evaluation of the mapping may result in an unbounded number of steps (rule applications). This distinction should be kept in mind during the development presented below.

3.2. Machine Emulation Results

We begin with some preliminary claims about the rule-based models and their use in emulating machines. In all cases the visualization models use the FB restrictions described above.

If the linear model can emulate a machine, so can the history model; if the history model can emulate a machine, so can the fixpoint model. This is obvious, since the history model can emulate the linear model simply by not referring to the previous spaces, while the fixpoint model can emulate the history model simply by not using any fixpoint rules. Naturally, the converse of this claim also holds; if the history model cannot emulate a machine, neither can the linear model, and if the fixpoint model cannot emulate a machine, neither can the history model.

The linear and history models are of equal computational power for the GSA problem. In GSA, we present a string and evaluate the mapping once, which in these models means we apply the rules once. This means that the history cannot be used (or rather, that the history spaces are constant for the one aapplication of the rules, so any predicate referring to these spaces is also a constant) and thus the history model reduces to the linear model.

If a particular model can emulate a NPDA with one stack, it can emulate a TM. Assume that a rule-based model emulates a NPDA with one stack; then it is storing some representation of the stack and examining and manipulating this representation in accord with the FB restrictions. This means that the model can also emulate a NPDA with two stacks and still be in accord with the restrictions, since (intuitively) we need to at most double the number of tuple types, rules, components of rule queries and productions, and so forth. However, a known result of automata theory is that a NPDA with two stacks can emulate a TM; thus, the rule-based model can emulate a TM.

Chomsky languages of Types 0, 1, and 2 are equivalent for any rule-based model, as are the corresponding automata (TM, linear-bounded TM, and NPDA). This is an immediate corollary of the previous claim. If a model can emulate a NPDA, it can emulate a TM; and if the model can emulate a TM it can emulate a NPDA or linear-bounded TM, from basic automata theory. Thus, for any rule-based model these three automata are equivalent—either all can be emulated, or none can. The equivalence of the languages follows from that of the automata.

As a result of the last claim, we only need to focus on the emulation of the FSM and the TM for the two problem instances. The following table summarizes the capabilities of each of the models, indicating for each whether or not that model is able to emulate the given machine when presented with the given problem instance:

Problem/Machine	Linear	History	Fixpoint
ISA/FSM	no	yes	yes
GSA/FSM	no	no	yes
ISA/TM	no	no	yes
GSA/TM	no	no	yes

Of course, the most significant aspect of this table is that it shows that the fixpoint model (even with the severe FB restrictions) is computationally equivalent to a TM, and by the Church-Turing thesis is therefore capable of performing any computation. Further, this holds even in the GSA case, meaning that a single mapping evaluation in the fixpoint model is capable of performing any computation.

Another aspect of the table deserves mention before we continue. Recall that we said that the ISA problem is similar to the manner in which data is presented to a FSM, while GSA is similar to the manner in which data is presented to a TM. Examining only the rows ISA/FSM and GSA/TM, we see that these two problems serve to distinguish the three rule-based models — the linear model cannot solve either, the history model can solve the first but not the second, and the fixpoint model can solve both. This result also demonstrates the fact that the three models have different computational power.

We now justify the entries in the table. The arguments vary depending on the entry. A "yes" entry is demonstrated by providing rules that allow the model to emulate the automaton. A "no" entry is justified by an argument as to why the emulation (or, equivalently, the language recognition problem) is not possible.

ISA/FSM and ISA/TM, linear. The argument here is based on the language aspect of the problem. Consider the situation when the linear mapping is processing the last character of the string; it must produce either *accept(true)* or *accept(false)*. These tuples in turn are produced by queries which ultimately depend only on the current state of the initial space, which contains a single *character(c)* tuple — in other words, the support or justification for generating the *accept* tuple can be based only on the last character of the string. The linear model is thus incapable of even distinguishing strings (except by their last character) and therefore cannot recognize either Type 3 or Type 0 languages—with the rather trivial exception of Type 3 languages in which all strings are a single character. From the automaton-emulation viewpoint, this amounts to the argument that the linear mapping cannot store information between successive applications and so has no means to keep track of the state of the FSM or TM.

GSA/FSM and GSA/TM, linear and history. The argument of the ISA case does not apply here, since the entire string is available for examination. In this case the problem is in the finite-bounded computational requirement, which requires that the number of distinct characters of the string that are examined during the application of the rules be bounded. However, Type 3 languages exist in which the number of characters that must be examined is unbounded — for example, the language over {**a**,**b**} described by the regular expression **a***, where each position in the string must be inspected to ensure that it is an **a**. Thus a linear mapping cannot solve the Type 3 GSA problem, which means it cannot solve the Type 0 GSA problem. By the previous equivalence claim, neither can a history mapping.

ISA/FSM, history. The emulation of a deterministic FSM is given in Figure 5. The FSM's state is represented by the *fsmstate* tuple. On each rule evaluation, the previous *fsmstate* and current *character* are used to generate the new *fsmstate* as well as an *accept* tuple; thus after each character is processed the emulation indicates whether the prefix of the string ending with that character is acceptable. This behavior could be circumvented if we assume that the end of the string is signaled in some fashion (e.g., the presence of a *done* tuple as well as the

A deterministic FSM is a 5-tuple $M = (Q, \Sigma, \delta, q_i, F)$ where
Q is a finite set of states,
Σ is a finite alphabet,
$q_i \in Q$ is the initial state of the automaton,
$F \subseteq Q$ is the set of accepting states of the automaton, and
$\delta : Q \times \Sigma \rightarrow Q$ is the transition function of the automaton.

$\langle\langle q,s\rangle,q'\rangle \in \delta$ means that if the automaton is in state q and reads symbol s from the tape, it will enter the state q'. ($\langle\langle q,s\rangle,q'\rangle \in \delta$ is equivalent to $q' = \delta(q,s)$.) We define the types *state* and *letter* to mean elements of the sets Q and Σ respectively.

The automaton M is translated into the following mapping:

```
mapping FSM_emulator
        input ⇒ output;

input space
        ⟨ letter s :: character(s) ⟩;

output space
        ⟨ state q :: fsmstate(q) ⟩;
        ⟨ boolean b :: accept(b) ⟩;

initialization
        fsmstate(qi);
        accept(qi ∈ F);
```

For each $\langle\langle q,s\rangle,q'\rangle \in \delta$, a rule of the form:

```
        rule name ≡
                        old.fsmstate(q),character(s)
                ⇒
                        fsmstate(q'),accept(q' ∈ F);
end
```

Figure 5. Emulation of a FSM in the history model.

character tuple), but it is not really undesirable. By inspection, we can see that the mapping meets all the FB requirements — note in particular that during execution there is always exactly one *fsmstate* and exactly one *accept* tuple.

ISA/TM, history. The argument is again based on the finite-bounded requirements. The total number of steps (rule applications) taken in the history model for an ISA problem is equal to the number of characters in the string. However, there exist TM-recognizable languages in which the number of steps cannot be so bounded. As an example, consider the language over {[,],(,)} which we informally describe as "balanced strings of these two types of parentheses, except each occurrence of] balances all open (up to the most recent open [" (this language is inspired by certain LISP interpreters). This language is actually Type 2 and can be recognized by a DPDA, but the number of steps required after processing a] character is unbounded — consider strings of the form [(((...(((], where we must "keep track" of the (symbols in case a) is encountered, but when the] is encountered we must discard the (unbounded) information about the (symbols.

GSA/TM, fixpoint. The emulation of a deterministic TM is given in Figure 6.We assume that the TM halts for every string (this property can be guaranteed by construction for Type 0 language acceptors, although of course the general halting problem is undecidable). The TM's configuration is represented by a *config(q,L,s,R)* tuple which indicates that the machine is in state *q* with the symbol *s* under the head, with the sequences *reverse(L)* on the tape to the left of the head and *R* to the right. Symbols outside the range so represented are assumed to be the "blank square" symbol. The reader may recognize this as the method whereby it is shown that a PDA with two stacks can emulate a TM, using the two stacks to store the contents of the tape. The *initiate* rule starts the machine in a configuration with the string written on the tape; the various fixpoint *emulate* rules mimic the way in which the TM changes from one configuration to another; and once the TM's operation halts, the fixpoint *preserve* rule maintains the configuration. Application of the fixpoint rules of the first sub-mapping thus stops once the emulated TM reaches a halt state. The next sub-mapping, containing only the *acceptor* rule, is then applied to produce the *accept* tuple. By inspection, the mapping satisfies the FB requirements.

ISA/TM, fixpoint. This problem can be solved by prepending a sub-mapping to the mapping of Figure 6. This sub-mapping collects the ISA *character* tuples into a list. When the last character is read (this must be signaled in some way) the list is presented as an *omega* tuple and the GSA solution applied to it. One additional finesse is required; the obvious FB way to create the list also reverses the string, so the emulated TM must be set up to recognize the reverse of the strings in the language. Given a TM which recognizes a language, construction of a TM which recognizes the reverse of the language is mechanical; in the emulation of Figure 6, it amounts to little more than exchanging *L* and *R* in the *config* tuples and **left** and **right** in the TM's transition function.

ISA/FSM and GSA/FSM, fixpoint. The ability of the fixpoint model to solve these problems is implied by the results for the corresponding TM problems.

To conclude this section, we note that removing the FB restrictions (in particular, permitting rules to use numbers and count symbols) increases the ability of the models to recognize certain types of languages. For example, we can recognize the Type 1 language $\{ a^n b^n c^n \mid n \geq 0 \}$ with a *linear* mapping in the GSA scheme, using the single rule[1]:

```
recognize ≡
        list of letter ω; integer len, third :
                omega(ω), len = | ω |, len mod 3 = 0, third = len/3,
                ⟨ ∀ integer i : 0 ≤ i < third :: ω[i] = a ⟩,
                ⟨ ∀ integer i : third ≤ i < 2*third :: ω[i] = b ⟩,
                ⟨ ∀ integer i : 2*third ≤ i < 3*third :: ω[i] = c ⟩
        ⇒
                accept(true);
```

[1] This rule uses the three-part notation ⟨ **op** quantified_variables : range_constraint :: expression ⟩ which is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range_constraint*. Each such instantiation of the variables is substituted in *expression* producing a multiset of values to which *op* is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range_constraint*, the value of the three-part expression is the identity element for *op*, e.g., *true* when *op* is ∀.

A deterministic TM is a 4-tuple $M = \langle Q, \Sigma, \delta, q_i \rangle$ where

Q is a finite set of states; Q_h is defined as $Q \cup \{$ ***halt-accept, halt-reject*** $\}$,

Σ is a finite alphabet; $\Sigma_\perp$ is defined as $\Sigma \cup \{ \perp \}$, where $\perp$ is the "blank tape" symbol,

$q_i \in Q$ is the initial state of the automaton, and

$\delta : Q \times \Sigma_\perp \rightarrow Q_h \times \Sigma_\perp \times \{$ ***left, right*** $\}$ is the transition function of the automaton.

$\langle\langle q,s\rangle,\langle q',s',d\rangle\rangle \in \delta$ means that if the TM is in state q with symbol s on the tape under its read head, it will enter state q', write the symbol s' on the tape, and then move the head one square in the direction d (left or right). We define the types *state* and *letter* to mean elements of the sets Q_h and $\Sigma_\perp$ respectively, and define the following two finite-bounded functions on lists:

hd(L) ≡ **if** L = [] **then** ⊥ **else** head(L);
tl(L) ≡ **if** L = [] **then** [] **else** tail(L);

The automaton M is then translated into the following two-stage mapping:

mapping TM_emulator
input ⇒ intermediate ⇒ output;

input space
⟨ list of letter ω :: omega(ω) ⟩;

intermediate space
⟨ state q; letter s; list of letter L, R :: config(q,L,s,R) ⟩;
Note: config(q,L,s,R) means the TM is in state q with s under the head, reverse(L) on the tape to the left of the head, and R on the tape to the right of the head; all other tape squares contain the blank symbol.

output space
⟨ boolean b :: accept(b) ⟩;

rule initiate ≡
list of letter ω : omega(ω)
⇒ config(q_i, [], hd(ω), tl(ω));

For each $\langle\langle q,s\rangle,\langle q',s',$ ***left*** *$\rangle\rangle \in \delta$, a rule of the form:*

fixpoint rule *name* ≡
list of $\Sigma_\perp$ L, R: old.config(q,L,s,R)
⇒ config(q', tl(L), hd(L), cons(s',R)) ;

For each $\langle\langle q,s\rangle,\langle q',s',$ ***right*** *$\rangle\rangle \in \delta$, a rule of the form:*

fixpoint rule *name* ≡
list of $\Sigma_\perp$ L, R: old.config(q,L,s,R)
⇒ config(q', cons(s',L), hd(R), tl(R)) ;

fixpoint rule preserve ≡
state q; letter s; list of letter L, R : old.config(q,L,s,R), (q = **halt-accept** ∨ q = **halt-reject**)
⇒ config(q,L,s,R);

rule acceptor ≡
state q; letter s; list of letter L, R : config(q,L,s,R)
⇒ accept(q = **halt-accept**);

Figure 6. Emulation of a TM in the fixpoint model.

Of course, evaluating the query portion of this rule requires considerable computation—in some sense, as much computation as recognizing the string using a fixpoint TM emulation. From that viewpoint, the only benefits of the above rule are that it simplifies the notation and is more understandable than would be the formulation in terms of a TM emulation.

Despite this, removing the FB restriction does *not* increase the computational power of the linear, history, and fixpoint mappings in terms of the Chomsky hierarchy. There are still Type 3 languages that cannot be recognized by a linear mapping (even in the GSA scheme), and there are still Type 0, 1, and 2 languages that cannot be recognized by a history mapping. Cases such as the above example should be considered as anomalies made possible by the use of numbers and unbounded query complexity.

4. Evaluation of Recursive Functions

The machine emulation and string acceptance results of the previous section provide a definitive theoretical characterization of the power of rule-based mappings. However, these results are somewhat divorced from the purpose of our mappings, viz., the construction of visualizations. In this section and the next we consider the use of the mapping models in program visualization. This section focuses on a specific problem which often arises in visualization, the evaluation of a certain type of recursive functions, while the next section has a more informal characterization in terms of the abstractive capabilities of the mappings.

We begin with an illustration of the class of functions which are of interest. Consider the problem of converting data organized as a tree (e.g., a parse tree, or a balanced binary tree used as a search tree) into a geometric tree as illustrated in Figure 7. The conversion requires the assignment of a coordinate in the final image to each of the abstract data nodes. In general, the underlying computation will not associate any geometric information such as coordinates with the data nodes (since the computation does not require this information), so the geometry must be synthesized from the available data. We may assume that this data includes the parent/child relationships among the abstract nodes, or that such relationships can be trivially extracted.

One of the geometric parameters which must be calculated is the Y-coordinate of each node. By inspection, this is equivalent to computing the distance of each node from the root of the tree, with conversion to the final geometry then simply a matter of scaling and translation. The distance is described by the recursive function:

$$distance(n) = \begin{cases} 0 & \text{if } n \text{ is a root} \\ distance(parent(n))+1 & \text{otherwise} \end{cases}$$

where *parent* is the relationship among the abstract nodes. This equation can also be viewed as defining *constraints* on the distances of the nodes. The roots are constrained to have a distance of 0, and each other node is constrained to have a distance one greater than that of its parent. Several visualization systems, including Tango and Animus [5], use such constraint-based specification of the relationships between objects; our results will thus allow a comparison of the capabilities of rule-based and constraint-based systems.

The calculation of this function can be cast this into the form of a rule-based mapping as follows. The input space contains three types of tuples: *node(x)* indicating that x is a node whose distance is to be computed; *root(x)* indicating that x is a root node of a tree; and *parent(x,y)* indicating that x's parent is y. We wish to produce an output space containing, for each *node(x)* tuple in the input space, a tuple of the type *distance(x,n)* indicating that node x is at a distance n from the root of its tree.

The evaluation of this function has two characteristics of interest. First, the number of examinations of the program data is unbounded since the "chain" from a node to the root is arbitrarily-long, and the evaluation requires one or more examinations of the data at each step. Second, these examinations must be performed in a particular order as determined by the chain of parent links (and thus, indirectly, by the results of the previous examinations). As a result, evaluation of this type of function requires unbounded serial computation.

For simplicity, we will focus on functions which have the general form of the *distance* function. That is, we are given a function $f : S \rightarrow V$, where S is some domain of interest and V is the range of the function. Also

node(0)	root(0)
node(1)	parent(1,4)
node(2)	parent(2,1)
node(3)	parent(3,8)
node(4)	parent(4,0)
node(5)	parent(5,8)
node(6)	parent(6,8)
node(7)	parent(7,4)
node(8)	parent(8,0)
node(9)	parent(9,1)

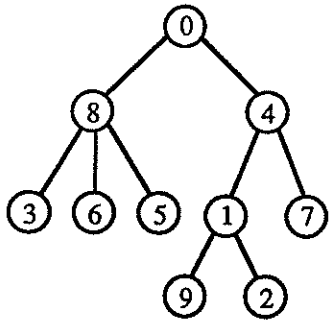


Figure 7. (left) A collection of facts about the state of a computation. (right) A representation of these facts in the form of a tree.

given is an input space and some set $X \subseteq S$. For each $x \in X$ we wish to produce a fact *function_name(s f(s))*. The function has the form:

$$f(s) = \begin{cases} b(s) & \text{if } s \in B \\ h(f(g(s))) & \text{otherwise} \end{cases} \quad (1)$$

Here $B \subseteq S$ and $b : B \rightarrow V$, $g : S \rightarrow S$, and $h : V \rightarrow V$. Thus, if $s \in B$ the value of $f(s)$ is $b(s)$; otherwise, we evaluate $g(s)$ to get another element of S, recursively compute $f(g(s))$ to get some value of V, then take $h(f(g(s)))$ as the value of $f(s)$. We assume that the test $s \in B$ and the functions b, g, and h can all be evaluated with only a finite and bounded number of examinations of the input space. In our *distance* example S is the set of abstract nodes, X the set of those x such that a *node(x)* tuple is present in the input space, V the set of naturals, f the *distance* function, B the set of root nodes (nodes with a *root(x)* tuple), b the constant function 0, g the *parent* function, and h the function such that $h(x) = x + 1$.

Although this form may seem to describe a rather limited class of functions, a surprisingly large number of the functions commonly encountered in visualization—particularly when synthesizing geometric information—have precisely this form. Further, all the results which we obtain are applicable to more complex functional forms.

In the next several sub-sections, we investigate each of the three mapping models and determine what types of recursive function they can evaluate. The results depend on two major factors. Most obviously, the nature of the function that is to be evaluated is important. As will be shown, although general recursive functions cannot be evaluated in the linear model, some special types of these functions can be evaluated in that model. A second observation applies to the use of these functions in the context of program visualization. When used in this manner, the input space is the state space of the underlying computation, and thus does not change in an arbitrary manner but according to the logic of that computation. It is often possible to exploit this property in the evaluation of the function.

4.1. Linear Model

As might be expected from the machine-emulation results, the linear model has the least ability to evaluate recursive functions. The linear model must perform the entire evaluation in a single application of the rules. Only a very limited class of functions can be so evaluated. The functions are those in which *all* needed inspections of the input space can be incorporated into a single query predicate[2], or in other words those in which the total number of queries of the input space is finite and bounded. By inspection of the general form in equation (1), this is possible if (and only if) the number of recursive evaluations of f is finite and bounded, that is, if the recursive depth of f is fixed and independent of the input space.

The technique for translating such functions into rules is similar to that of unrolling a loop in traditional programming languages. The recursive portion of equation (1) is simply expanded as often as required by the recursive depth of the function. This will generally result in several queries: (at least) one for the base case(s), and (at least) one for each depth in the recursion. The value of the function is then computed by application of the b and h functions.

Returning to the tree example, assume that we know (perhaps by examination of the correctness properties of the program) that all nodes in the tree will have distance less than or equal to three. Then we can "unroll" the recursive definition of the *distance* function to produce the four rules of Figure 8. The first rule is the basis case and computes the height of the root node. The second rule computes the height of the children of the root, the third the height of the grandchildren, and the last the height of the great-grandchildren. Note that each of these rules has the form *"node(n), ... root(n')"*, where *node(n)* represents the determination that the height of n must be computed, the *"..."* portion is the unrolled recursive inspection of the input space, and *root(n')* is the base case. Also note that the computed height is found by applying the functions in the original recursive definition, e.g., the number 3 in the last rule is actually obtained from $h(h(h(b(root))))$, where $h(x) = x + 1$ and $b(root) = 0$.

---

[2] A multiple-stage mapping would allow the computation to be spread over several stages; however, as indicated earlier, all such mappings have an equivalent single-stage mapping in the linear model.

```
rule root_height ≡
        integer x :
              node(x), root(x)
        ⇒
              distance(x,0);

rule child_height ≡
        integer x, px :
              node(x), parent(x,px), root(px)
        ⇒
              distance(x,1);

rule grandchild_height ≡
        integer x, px, gpx :
              node(x), parent(x,px), parent(px,gpx), root(gpx)
        ⇒
              distance(x,2);

rule greatgrandchild_height ≡
        integer x, px, gpx, ggpx :
              node(x), parent(x,px), parent(px,gpx), parent(gpx,ggpx), root(ggpx)
        ⇒
              distance(x,3);
```

Figure 8. Linear-model rules for computing *distance(s)* when it is known that the maximum height of the trees is 3.

4.2. History Model

The ability of the history model to evaluate recursive functions is equivalent to that of the linear model if we know nothing about the way in which the input space changes. Intuitively, this is because both models use the same basic rule-evaluation strategy, that is, a single application of the rules. The only advantage the history model has is the ability to examine the previous instances of the input and output spaces, which might include function values that were computed in the previous step. However, with no knowledge of the ways in which the input space may be modified we cannot guarantee that the previously-computed function values remain valid.

We again return to the tree-height example for an illustration. Assume that the input space contains the tuples *node(1), node(2), parent(2,1), root(1).* Then the output space contains the facts *distance(1,0)* and *distance(2,1).* Assume now that the input space changes so that we must re-evaluate the function. In the history model, we now have access to the facts *distance(1,0)* and *distance(2,1)* from the first application of the rules, but these tuples are of no aid in evaluating the function since the input space may have changed arbitrarily—for example, it may now contain the tuples *node(1), node(2), parent(1,2), root(2),* and neither of the previous function values are valid.

Now assume that we know some information about the operation of the underlying computation; specifically, it starts with only the root node(s) present, and thereafter the tree structure is altered only by the addition and removal of single leaves. This can be more formally expressed by saying that the initial state contains only *node(x), root(x)* for some node(s) *x*, and that the only allowed changes are 1) adding a leaf by the addition of the tuples *node(x), parent(x,y)* to a state which does contains a *node(y)* tuple but not a *node(x)* tuple, 2) removing a non-root leaf by removal of the tuples *node(x), parent(x,y)* from a state which does not contain any tuples of the form *parent(z,x)*, or 3) removing a root by removal of the tuples *node(x), root(x)* from a state which does not contain any tuples of the form *parent(z,x).* From this behavior, we can deduce the following:

1) If the current input space contains *root(x)*, the current output space should contain *distance(x,0).*

2) If both the current and previous input spaces contain *node(x)*, then the previous output space contains a *distance(x,n)* tuple and the current output space should contain a *distance(x,n)* tuple.

3) If the current input space contains *node(x)* and *parent(x,y)* but the previous input space does not, then the previous output space contains a *distance(y,n)* tuple and the current output space should contain a *distance(x,n+1)* tuple.

These conclusions allow us to formulate the rules in Figure 9 to compute the *distance* tuples for the computation.

Note that the "unrolling" technique used with the linear model is also applicable for the evaluation of the third set. If for example the underlying computation could in a single step add a node *p* as the child of a (previously-present) node *gp* and also add a node *x* as the child of *p*, it would still remain possible to compute the *distance* function since we could construct the rule:

```
compute_grandchild_height ≡
    integer x, p, gp, n :
        node(x), ¬old.node(x), parent(x,px), parent(px,gp), old.distance(gp,n)
    ⇒
        distance(x,n+2);
```

This observation clarifies the technique involved in evaluating the function using the history model. For each $x \in S$, we must show that evaluation of $f(x)$ involves only a finite and bounded number of recursive evaluations, just as in the linear model. However, in the history model we are allowed to take as "basis" elements of the recursion any elements whose function value was found in the previous step, provided we can show that the calculated value remains valid.

We can formalize this as follows. First, demonstrate that the recursive function $f: S \rightarrow V$ in conjunction with the known behavior of the underlying computation has the properties, for some $C \subseteq S$:

- For all $s \in C$, $f(s)$ was evaluated in the previous step and has not been altered by the state change.

```
rule root_height ≡
        integer x :
                node(x), root(x)
        ⇒
                distance(x,0);

rule retain_height ≡
        integer x, n:
                node(x), old.distance(x,n)
        ⇒
                distance(x,n);

rule compute_height ≡
        integer x, px, n :
                node(x), ¬old.node(x), parent(x,px), old.distance(px,n)
        ⇒
                distance(x,n+1);
```

Figure 9. History-model rules for computing *distance(s)* when it is known that the trees initially consist only of the root node and thereafter are modified only by the addition or removal of single leaves.

- For all $s \notin C$, in the evaluation of $f(s)$ either a basis element or some element of the set C is reached in a finite and bounded number of recursive evaluations of f.

Once this is shown, rules can be easily constructed. The first type of rule computes $f(s)$ for $s \in C$ as exemplified by *retain_height* in Figure 9. The query of this type of rule consists first of a test to determine if $s \in C$, followed by an examination of the previous output space to find the value of the function. The second type of rule computes $f(s)$ for $s \notin C$, using constructions similar to those of the linear model with the addition of the test to determine that $s \notin C$. The rules *root_height* and *compute_height* of Figure 9 are of this type, as is the *compute_grandchild_height* rule given above.

Our experiences indicate that this history-based evaluation technique is not widely applicable, since functions (and underlying computations) with these characteristics are not very common. Of the several dozen programs for which we have constructed visualizations, such situations were encountered only twice. Thus, for all practical purposes the linear and history models are equivalent in their ability to evaluate recursive functions.

4.3. Fixpoint Model

The fixpoint model is capable of evaluating any recursive function in a single rule-evaluation, even if we have no knowledge of the behavior of the underlying computation. This result is an obvious consequence of the previous demonstration that the fixpoint model is Turing-equivalent.

Figure 10 shows the fixpoint rules for computing the node distances in our tree example. Note that these rules correspond to the recurrence relation that defines the *distance* function—the first rule is the basis case, and the second the recursive case. The development of the rules for this problem is simplified in that if we need *distance(x)* for any node x we also need to compute *distance(y)* for any ancestor y of x. Thus, we can perform the computation "top-down", by first computing the height of the roots with the first (basis) rule, then of the children of the roots with the second (recurrence) rule, then the grandchildren, and so forth. The total number of rule applications to reach fixpoint is thus one more than the maximum distance of any node.

In the general case, we cannot use this approach since there is no guarantee that if $f(x)$ must be computed so must $f(g(x))$, the term required by the recurrence of equation (1). For such systems, we must ensure that the need to compute $f(x)$ initiates the computation of $f(g(x))$. We illustrate the general technique with the rules in Figure 11. Here, we assume that only the heights of some subset of the nodes are required; these nodes, and only these nodes, have a *node(x)* tuple in the input space. The *root* and *parent* tuples for the entire tree are still present in the input space.

Computation of the distance of a node x is triggered if: 1) either a *node(x)* tuple is present in the input space or a *compute_distance(x)* tuple is in the previous output space, and 2) the distance of node x is not known (not in the previous output space). This is reflected in the queries of the first three rules. The first rule, *root_height*, computes the values for the basis case. The recursive case is represented by two rules. The first computes *distance(x)* when the distance of the parent node is known, as in Figure 9. The second detects the case where *distance(x)* is needed but *distance(parent(x))* is not yet known, and initiates computation of the latter value by producing a *compute_distance* tuple for the parent. Finally, the rule *retain_heights* ensures that if the distance of a node is required by the presence of a *node(x)* tuple in the input space, the height will be retained once it is computed.

When the rules of Figure 11 are applied, the final fixpoint output space will contain *distance(x,n)* tuples for exactly those x such that the input space contains a *node(x)* tuple. The computation initiated by a node "travels" up the tree generating a *compute_distance* for each ancestor until the root is reached, then travels back down generating the distances. The total number of rule applications to reach fixpoint is one plus twice the maximum computed distance.

We conclude with a comment on the efficiency of rule-evaluation and of fixpoint evaluation in particular. The number of rule-applications cited above may seem somewhat alarming. Surely, it might be said, a more traditional imperative form of function evaluation would be more efficient as well as more understandable. To an extent this is true, and we have plans to add recursive functions to Pavane. However, we cite two factors that indicate that the rule-based forms and the traditional recursive functions may actually be of comparable efficiency. First, the number of rule-evaluations is proportional to the depth of the recursion, and in fact when rules such as 11 are used the number is approximately twice this depth. Similarly, the number of inspections of the state space that

```
fixpoint rule root_height ≡
        integer x :
                node(x), root(x)
        ⇒
                distance(x,0);

fixpoint rule compute_height ≡
        integer x, px, n :
                node(x), parent(x,px), old.distance(px,n)
        ⇒
                distance(x,n+1);
```

Figure 10. Fixpoint-model rules for computing *distance(s)*, applicable for any behavior of the underlying computation. These rules compute the distances of all the nodes in the tree in a "top-down" fashion, starting with the root nodes.

```
fixpoint rule root_height ≡
        integer x :
                (node(x) ∨ old.compute_distance(x)), ¬⟨∃ integer n :: old.distance(x,n)⟩,
                root(x)
        ⇒
                distance(x,0);

fixpoint rule compute_height1 ≡
        integer x, px, n :
                (node(x) ∨ old.compute_distance(x)), ¬⟨∃ integer n :: old.distance(x,n)⟩,
                parent(x,px), old.distance(px,n)
        ⇒
                distance(x,n+1);

fixpoint rule compute_height2 ≡
        integer x, px :
                (node(x) ∨ old.compute_distance(x)), ¬⟨∃ integer n :: old.distance(x,n)⟩,
                parent(x,px), ¬⟨∃ integer n :: old.distance(px,n)⟩
        ⇒
                compute_distance(px);

fixpoint rule retain_heights ≡
        integer x, d :
                node(x), old.distance(x,d)
        ⇒
                distance(x,d);
```

Figure 11. Fixpoint-model rules for computing *distance(s)*. These rules can be used even if only the heights of a subset of the nodes of the tree are desired; the heights for the required nodes are computed in a "bottom-up" fashion starting from the nodes.

are performed in the rule-based case is at most twice that of the recursive case. Our second point, which is perhaps the more interesting, is that the fixpoint evaluation computes all the needed values *in parallel*, and the evaluation process itself is highly suited to parallel implementation. The total execution time of such a parallel implementation of a fixpoint mapping would probably compare favorably with, or even improve on, that of a serial implementation that used traditional recursive functions. Unfortunately at this time we do not have a parallel version of Pavane and so cannot provide any concrete figures.

5. Abstractive Capabilities

We have used our view of visualization as a mapping of information to graphical form as the basis for a taxonomy of visualization systems [8]. One component of our taxonomy is the level of abstraction that a system supports, by which we mean the type of transformation performed by the visualization. We can distinguish three levels of abstraction based on the form of the transformation:

- *Direct* abstractions are one-to-one mappings of program data to the image. The graphical components of the final image correspond exactly to the selected data. Such visualizations are simple to construct (the process can often be automated) and effective in many contexts, but in many cases are insufficient to convey an understanding of the program's operation.

- *Structural* abstractions are also mappings of program data to the image, but are not one-to-one. Structural abstractions may conceal or encapsulate some information, as by depicting a complex data structure as a single graphical object; this can be viewed as a many-to-one mapping. Alternatively, the abstraction may structure the image to emphasize portions of the data. The latter might be viewed as a one-to-many mapping (in that a single data object might be represented in several different ways).

- *Synthesized* abstractions are mappings in which the domain is not simply the program data but also some information that must be generated by the visualization. Such abstractions arise in several ways, with one of the most common being the need to generate geometric information such as the node coordinates in the previous section. Another common case arises when historical information must be presented; the program does not normally maintain information about its previous activities, so the visualization must record the needed data for later use.

We will also discuss a second aspect of the visual presentation of information. Often, a program visualization is more effective if instead of simply showing discrete images corresponding to the succession of states, a smoothly-animated interpolation between the images is shown. This can, for example, allow the viewer to better track the state changes that occur. In addition, animation permits the use of a variety of visual events—color flashing, size changes, and so forth—that can also assist the viewer by focusing attention on critical elements of the image. We refer to the use of animation and other visual events as *explanatory* presentation of information. In our taxonomy, this component is orthogonal to the abstraction axis in that it can be used with direct, structural, or synthesized abstractions.

In the remainder of this section we discuss the ability of each of the three mapping models to perform the above types of abstraction and presentation. The discussion is largely informal. This is partly due to the somewhat informal nature of the concept of abstraction, but we are also able to exploit the concrete results of the previous sections to illustrate our points.

Linear mapping. The linear mapping is capable of performing direct and structural abstractions, since these types of abstractions are simply mappings from the current state to the image, with no synthesis of information. (From this result, we know that the history and fixpoint mappings are also able to construct direct and structural abstractions, since in section 3 we showed that these mappings are more powerful than the linear mapping.) However, for general problem instances the linear mapping cannot perform synthesized abstractions, nor can it produce explanatory presentations, as shown here.

A synthesized abstraction involves the production of information which is not contained in the current state of the underlying computation. This information may be from previous states of the computation, for example when we wish to present the sequence of operations that a computation has performed. However, because the linear model has no history capability, so cannot store such information. The synthesized information may also be

computable from the current state, as in the tree *distance* example of the previous section, but again these functions cannot in general be computed in the linear model.

The lack of any history capability also generally makes the production of explanatory presentations impossible, since such presentations require knowledge of historical information—in this case, the most recent output of the visualization. Of course, in certain specialized cases it may be possible to deduce the needed information. As a simple example, consider a bubble sort in which the state consists of two index variables and an array of numbers. A direct animation can examine this state and determine if the array elements *will be* exchanged; it can then produce an animation of the exchange *before* the underlying computation actually performs the swap. Note, however, that this is the reverse the normal method of constructing such animations, in which a state change is observed and only then presented.

History mapping. As indicated above, the history mapping is capable of performing direct and structural abstractions. It is also able to perform limited types of synthesized mappings and explanatory presentations. The types of synthesized mappings which can be implemented are those in which historical information about the computation must be accumulated by the visualization and presented. We can, for example, show the sequence of messages that have been transmitted over each link in a message-passing system, or the exchanges of a sorting algorithm. However, we still cannot synthesize arbitrary information, since (as shown previously) the history mapping cannot compute arbitrary recursive functions.

The history mapping is able to construct many explanatory presentations which involve smooth interpolation between successive states of the underlying computation. This is because it can record the graphical objects produced by each evaluation of the mapping and, by examining the current and previous collections of objects, construct the needed interpolation. (Indeed, this ability to interpolate between images originally prompted the use of a one-step history.) Although the history mapping cannot construct arbitrary explanatory presentations, it is powerful enough to produce most of the commonly-used effects.

As an example of the type of animation which cannot be produced, we turn to the Floyd-Warshall shortest-path algorithm. This algorithm computes, in parallel, the distances (sum of edge weights) between all nodes in a graph. The computation operates on a matrix of distances between nodes, proceeding by a process of "scanning" nodes. An important invariant property of this program can be informally stated as, "At all times the distance between any two nodes is that of the shortest path between the nodes all of whose internal nodes are scanned."

Figure 12 illustrates a possible visualization of this algorithm. The graph at the left is the input to the algorithm. The other two graphs show the scanned nodes (depicted in black) and the currently-known paths from node 0 (depicted as solid lines). The center graph shows the computation's state after nodes 0 through 4 have been scanned; note that only distances and paths to nodes 5 and 8 are known, since by the program invariant all the internal nodes of the paths must be scanned. The graph on the right shows the state after node 5 is scanned. Paths (though not necessarily shortest paths) are now known to all the other nodes. (As an aside, the Floyd-Warshall algorithm does not maintain this path information, but it can be easily synthesized from the distance matrix, even with a linear mapping.)

Assume now that we want to animate the transition between these states by having the new paths "grow" outward from the scanned node (number 5). That is, from node 5 paths will first extend to nodes 4 and 6, taking (say) five Pavane frames to reach these nodes. When the path reaches node 4, it will continue to extend to node 1 (again taking five frames), then from there to nodes 7 and 3, and so forth as shown by the arrows in the right graph in Figure 12. In order to produce this effect, we need to determine the times (frame numbers) at which the growth of each line begins and ends. However, determining these times is equivalent to computing the *distance* function of the previous section, where distance is measured from the scanned node along the shortest paths. As we have seen, this function cannot be computed with a history mapping.

Fixpoint mapping. The fixpoint mapping is capable of performing any abstraction and constructing arbitrarily-complex explanatory presentations. This is an immediate consequence of its Turing-equivalence as shown in section 3. For example, the fixpoint mapping can compute the animation times for the Floyd-Warshall algorithm described above. The required rules are similar to those of Figure [FIX-TREE], reflecting the close relationship between the two functions.

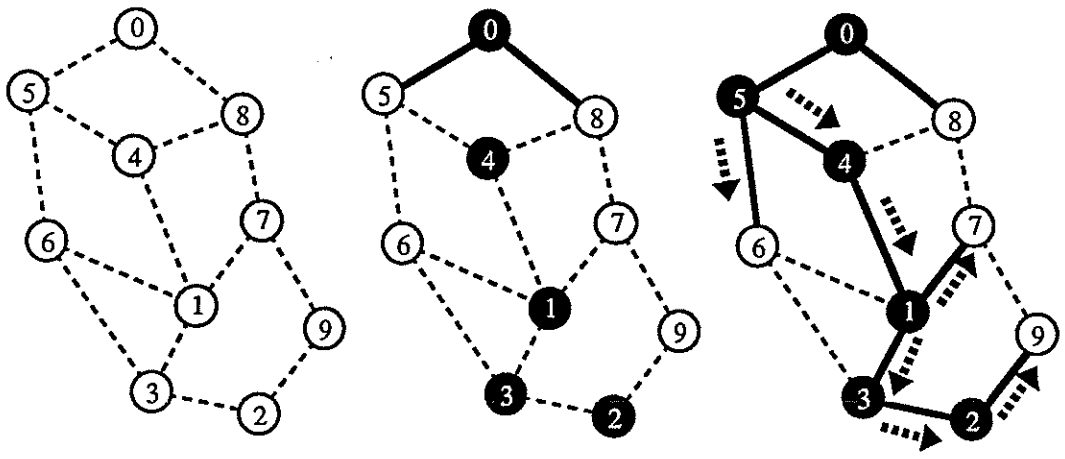


Figure 12. Illustration of path information in the Floyd-Warshall algorithm. The left graph is that used as input to the algorithm. The center graph shows the paths (solid lines) from node 0 after nodes 0 through 4 have been scanned. The right graph shows the paths after node 5 is scanned, with arrows showing the desired animation of the growth of the new paths. Although these images could be produced with a linear mapping, producing the animation as described in the text requires a fixpoint mapping.

6. Summary and Conclusions

The work presented in this paper was originally motivated by the simple question, "How can we use Pavane to construct the types of visualizations that we want?" By answering this question, and more importantly by expanding the capabilities of Pavane to allow us to produce the effects that we desired, we have gained a better understanding of Pavane, of similar rule-based systems, and of declarative systems in general. One of the most significant results was the discovery that the three mapping models used in Pavane—linear, history, and fixpoint—are of distinct computational power. The most important results of this evaluation are summarized in Table 1.

Although our results are presented using a rule-based model, we believe that they are also applicable to any declarative system which transforms program states to images; this belief is justified primarily by the correspondences between our three models and the finite automata. We hypothesize the following correspondences between our models of rule-application and other declarative models as follows:

- The linear model has the same capabilities as any declarative model that cannot store information and permits only a bounded amount of computation during the production of each image.
- The history model has the same capabilities as any declarative model that can store information and permits only a bounded amount of computation during the production of each image.
- The fixpoint model has the same capabilities as any declarative model that can store information and permits unbounded computation during the production of each image.

Investigation of other declarative systems to either confirm or deny this hypothesis is indicated. Such research should probably begin with a characterization of the power of the system in terms of the Chomsky hierarchy, which would permit immediate comparison with our own results.

		Linear	History	Fixpoint
Machine Emulation	FSM (ISA)		✓	✓
	TM (GSA)			✓
Recursive Function Evaluation	Fixed-depth	✓	✓	✓
	Other		some	✓
Abstraction	Direct/structural	✓	✓	✓
	Synthesized			✓
	Explanatory		some	✓

Table 1. Summary of the results presented in this paper. The notation "some" indicates that only a restricted subset of the problem, as identified in the text, can be solved using the model.

Acknowledgments: This work was supported in part by the National Science Foundation under the Grant CCR-9015677. The government has certain rights in this material.

References

[1] Brown, M. H., "Exploring Algorithms using Balsa-II," *IEEE Computer*, vol. 21, no. 5, pp. 14-36, 1988.

[2] Brown, M. H., "Zeus: A System for Algorithm Animation and Multi-View Editing," *1991 IEEE Workshop on Visual Languages*, IEEE Computer Society Press, Kobe, Japan, pp. 4-9, 1991.

[3] Forgy, C. L., "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17-37, 1982.

[4] Helttula, E., Hyrskykari, A., and Raiha, K.-J., "Graphical Specification of Algorithm Animations with ALADDIN," in *Proceedings of the 22nd Annual Conference on Systems Sciences*, pp. 892-901, 1988.

[5] London, R. L., and Duisberg, R. A., "Animating Programs Using Smalltalk," *IEEE Computer*, vol. 18, no. 8, pp. 61-71, 1985.

[6] Roman, G.-C., and Cox, K. C., "A Declarative Approach to Visualizing Concurrent Computations," *IEEE Computer*, vol 22, no. 10, pp 25-36, 1989.

[7] Roman, G.-C., Cox, K. C., Wilcox, C. D., and Plun, J. Y., "Pavane: a System for Declarative Visualization of Concurrent Computations," *Journal of Visual Languages and Computing*, vol. 3, no. 1, pp. 161-193, 1992.

[8] Roman, G.-C., and Cox, K. C., "Program Visualization: The Art of Mapping Programs to Pictures," in *Proceedings of the 14th International Conference on Software Engineering*, pp. 412-420, 1992. (A revised version of this paper, titled "A Taxonomy of Program Visualization Systems", is currently in submission to *IEEE Computer*.)

[9] Stasko, J., "TANGO: A Framework and System for Algorithm Animation," *IEEE Computer*, vol 23, no. 9, pp. 27-39, 1990.